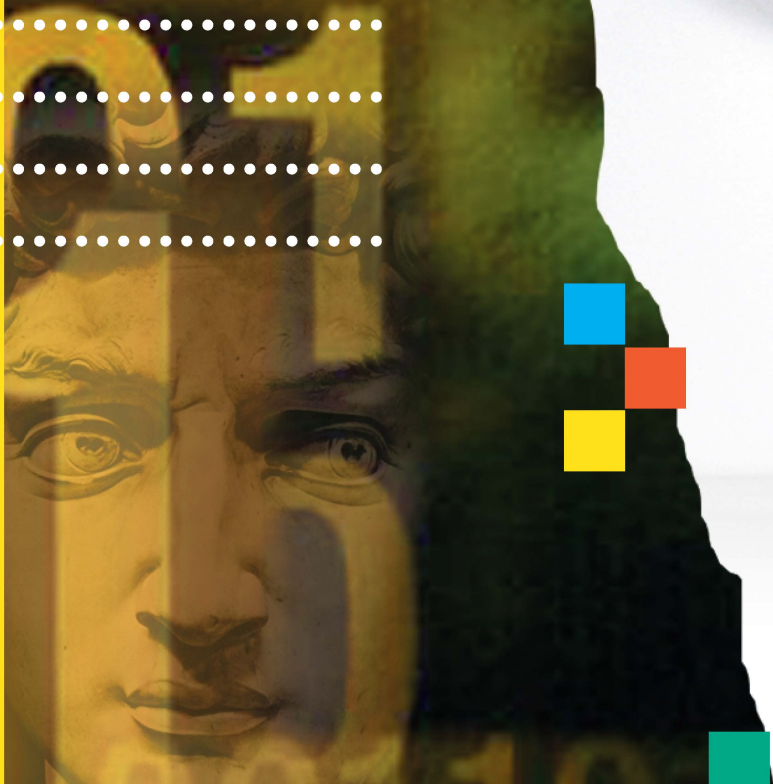


it courseware™

TRAINING MATERIALS FOR IT PROFESSIONALS

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited



EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

This material is copyrighted by LearningPatterns Inc. This content and shall not be reproduced, edited, or distributed, in hard copy or soft copy format, without express written consent of LearningPatterns Inc. Copyright © LearningPatterns Inc.

For more information about Java Enterprise Java, or related courseware, please contact us. Our courses are available globally for license, customization and/or purchase.

LearningPatterns. Inc.

Services@learningpatterns.com | www.learningpatterns.com

Global Courseware Services

982 Main St. Ste. 4-167| Fishkill, NY 12524 USA
212.487.9064 voice and fax

Java, and all Java-based trademarks and logo trademarks are registered trademarks of Oracle, Inc., in the United States and other countries. LearningPatterns and its logos are trademarks of LearningPatterns Inc. All other products referenced herein are trademarks of their respective holders.

Table of Contents: Intermediate/Advanced Java (11+)

<i>Intermediate/Advanced Java (11+)</i> _____	1
Course Overview _____	2
Course Overview _____	3
Course Agenda _____	4
Course Agenda _____	5
Review Sessions – Guidelines _____	6
Typographic Conventions _____	7
Labs _____	8
<i>Preface: Java State of the Union</i> _____	9
New Java Release Cycle _____	10
Versions and Support – Illustrated _____	11
What's In _____	12
What's Out _____	13
What We'll Cover _____	14
Thinking in Java 9+ _____	15
<i>Session 1: Review Basics</i> _____	16
Session Objectives _____	17
Java Environment _____	18
Java – a Language and a Platform _____	19
Lifecycle of a Java Program – Illustrated _____	20
Java Language vs. Java Runtime _____	21
Applications, Deployments, and Libraries _____	22
Lab 1.1 – Getting Started _____	23
Classes and Objects _____	24
Classes and Objects _____	25
Instance Variables (or Fields) _____	26
Creating Objects _____	27
Methods _____	28
OO Principle – Cohesion _____	29
OO Principle – Data Encapsulation _____	30
Data Encapsulation – Example _____	31
Creating and Using Objects – Example _____	32
Constructors _____	33
Writing Constructors _____	34
Special Constructor Rule _____	35
Removing Constructor Redundancy _____	36
Sharing Constructor Code – <code>this()</code> _____	37
Object Contracts – <code>toString()</code> _____	38
Composition and Delegation _____	39
Lab 1.2 – Classes and Objects – Part One _____	40
Java Naming Conventions – Review _____	41
JavaBeans – Naming Conventions As Law _____	42
JavaBeans Provides a Useful Abstraction _____	43
JavaBeans Caveats _____	44
Wrapper Classes _____	45
Autoboxing _____	46
Optional<T> _____	47

Working with <code>Optionals</code>	48
Object Contracts – <code>equals()</code>	49
Custom <code>equals()</code> Method – Example	50
Object Contracts – <code>hashCode()</code>	51
Custom <code>hashCode()</code> Method – Example	52
Importance of <code>hashCode()</code>	53
Sorting Objects – <code>Comparable</code>	54
<code>Comparable</code> Contract and Example	55
<code>Comparable</code> – Additional Examples and Tips	56
Alternative Sorting – <code>Comparator</code>	57
Sorting in Action – Example	58
Lab 1.3 – Classes and Objects – Part Two	59
String Class – Additional Details	60
StringBuffer / StringBuilder	61
Regular Expressions	62
Static Fields	63
Static Methods	64
Accessing Static Members	65
Static Fields – Demonstrated	66
Packages, Enums, Arrays	67
Packages	68
Organizing Files and Packages	69
Imports	70
Visibility – Fields and Methods	71
Static Imports	72
Enums	73
Enums – Additional Details	74
Lab 1.4 – Packages and Enums	75
Arrays	76
Working with Arrays	77
Array Iteration	78
Varargs Parameters	79
Lab 1.5 – Conditionals, Arrays, Varargs	80
Exceptions	81
Overview of Exceptions	82
Checked vs. Unchecked Exceptions	83
The Throwing Side	84
The Catching Side	85
Catching Multiple Exceptions	86
Multicatch	87
<code>finally</code>	88
<code>try-with-resources</code>	89
Exception Chaining	90
Exception Chaining – Example	91
Exception Handling – Best Practices	92
Lab 1.6 – Exceptions	93
Date and Time API	94
Previous Java Date/Time Support – History	95
Java 8+ Date/Time Support – Overview	96
Date and Time Classes – Overview	97
Creating Dates and Times	98
Parsing Dates and Times	99
Formatting Dates and Times	100
Accessing Date and Time Fields	101

Deriving New Values by Field	102
Deriving New Values by Adjustment	103
Adding and Subtracting Time	104
Period and Duration – Intervals of Time	105
Interval between Two Dates/Times	106
Adding an Interval to a Date/Time	107
Instant	108
Time-Zones	109
Equivalent Local Time	110
Lab 1.7 – Working with Dates and Times	111
New Language Features	112
- Java Modules -	113
Introducing Java Modules	114
Modular Encapsulation – Illustrated	115
Module Name vs. Package Name	116
Modules and Directory Structure	117
Modular Projects and the Module Path	118
- Type Inference: <code>var</code> Comes to Java -	119
Using Local-Variable Type Inference	120
- Switch Expressions -	121
<code>switch</code> As Statement vs. Expression	122
The New <code>switch</code> – Example	123
Working with Preview Features	124
Session Review	125
Session Review	126
Session 2: Review Inheritance and Interfaces	127
Session Objectives	128
UML Overview	129
UML Defined	130
Class Diagrams	131
Sequence Diagrams	132
UML Diagrams – Use Your Hands	133
UML in the IDE	134
Inheritance	135
Inheritance Defined	136
Inheritance – Illustrated in UML	137
Inheritance – in Code	138
What Gets Inherited?	139
Overriding Methods – Details	140
Overriding Methods – <code>@Override</code>	141
Lab 2.1 – Inheritance	142
Polymorphism – Illustrated	143
Important OO Principles	144
Constructor Chaining with <code>super()</code>	145
Constructor Chaining in Action	146
Object Construction in Memory	147
Inheritance and Static Methods	148
Interfaces	149
Interfaces Defined	150
Implementing an Interface	151
Interface Types	152
Interfaces in UML – Another Form of IS-A	153

OO Principles Still Apply	154
Specifications Use Interfaces	155
Interface Inheritance	156
Implementing Multiple Interfaces	157
Lab 2.2 – Interfaces	158
New Interface Features	159
Default Methods in Interfaces	160
Default Methods – Example	161
Motivation and Benefits of Default Methods	162
Motivation and Benefits of Default Methods	163
Inheritance Issues	164
Static Methods in Interfaces	165
Benefits of Static Methods	166
Private Methods in Interfaces	167
Functional Interface – Defined	168
Functional Interfaces – Details	169
Case Study – Comparator	170
Lab 2.3 – Default and Static Interface Methods	171
Guidelines	172
Commonality and Variability	173
When to Use an Abstract Class	174
When to Use an Interface	175
Concrete Class, Abstract Class, or Interface?	176
OO Principle – Coupling	177
Session Review	178
Session Review	179
Session 3: JUnit	180
Session Objectives	181
Overview	182
Unit Testing Overview	183
JUnit Overview	184
JUnit 4 vs. JUnit 5	185
Writing a Test – First Example	186
Running Tests in the IDE	187
Running Tests in Other Environments	188
Naming Conventions and Organizing Tests	189
Lab 3.1 – Getting Started with JUnit	190
Tests, Assertions, and Fixtures	191
Writing Test Methods	192
Assertions	193
Assertions Are Everything	194
Test Fixtures – @Before and @After	195
Test Fixtures – Example	196
Test Fixtures – @BeforeClass and @AfterClass	197
Order of Execution	198
Lab 3.2 – Assertions and Fixtures	199
Testing for Exceptions	200
Verifying the Exception	201
Test Suites	202
Alternate Test Runners – @RunWith	203
Lab 3.3 – More Practice with JUnit	204
Best Practices and TDD Overview [Read for Homework]	205

Tests Are Real Code	206
What to Test	207
Testing <code>void</code> and Private Methods	208
Be Thorough	209
Many Small Tests vs. Fewer Large Tests	210
Try to Break It	211
Narrow and Broad Testing	212
Characteristics of Good Tests	213
Testing Anti-Patterns	214
Test-Driven Development (TDD) – Overview	215
Test-Driven Development (TDD) – Overview	216
TDD Takes Practice	217
Further Reading	218
Lab 3.4 – Practicing Best Practices [Optional Homework Lab]	219
Lab 3.5 – Test Run Alternatives [Optional Homework Lab]	220
Session Review	221
Session 4: Collections and Generics	222
Session Objectives	223
Collections Overview	224
Collections Framework	225
Collections Hierarchy – Illustrated	226
Collection Interface	227
Generics and Type-Safe Collections	228
Quick Aside: Collections of Object	229
Be General	230
Diamond Operator	231
Lists, Sets, and Maps	232
Interfaces and Contracts: List Set Map	233
List<E> Interface	234
Using List<E> – Example	235
Set<E> Interface	236
Map<K, V> Interface	237
Using Map<K, V> – Example	238
Iterating over Collections and Maps	239
Autoboxing with Collections	240
Summarizing Collection Features	241
Consistent with Equals	242
Lab 4.1 – Using Collections	243
Utility Classes – Collections and Arrays	244
Collections Class	245
Arrays Class	246
Collection Factory Methods	247
Collection Factory Methods – Examples	248
Writing Generic Classes	249
Generics in Collections – Recap	250
Generics in General – Deeper Understanding	251
Inheritance with Generic Types	252
Inheritance with Generic Types	253
Wildcard Parameter Types – Why?	254
Wildcard Parameter Types	255
Wildcard Parameter Types – Examples	256
Wildcard Parameter Types – Examples	257
Generic Methods	258

Writing Generic Classes	259
Lab 4.2 – Writing Generic Classes	260
Type Erasure	261
Session Review	262
Session 5: Techniques of Object Creation	263
Session Objectives	264
Design Patterns Overview	265
Design Patterns – Catalog of Experience	266
Design Patterns Overview	267
The Gang of Four (GoF)	268
Design Patterns Vocabulary	269
Controlling Object Creation	270
Controlling Object Creation	271
Limitations of new Operator	272
Alternative Techniques – Features	273
Alternative Techniques – Features	274
Common Characteristics	275
Singleton Pattern	276
Singleton Pattern – Definition and Uses	277
Don't Make It Hard	278
Aside: When Do Classes Get Loaded?	279
General Technique	280
Lazy Initialization	281
Concurrency Issues	282
Thread-Safe Lazy Initialization	283
Singleton via Enum	284
Lab 5.1 – Examining Singletons	285
Exceptions during Creation	286
Subclassing Singletons	287
Container Alternatives	288
Simple Factory	289
Simple Factory – Illustrated	290
Factory Implementation – Example	291
Coupling in the Client	292
Coupling in the Factory	293
Lab 5.2 – Simple Factory	294
Factory Method Pattern	295
Factory Method Pattern – Description	296
Hierarchies of Factories and Products	297
Some Patterns Terminology	298
Example – Collection and Iterator	299
Applicability	300
Java EE Example – JPA	301
JPA Example in Factory Method UML	302
Implementation Options	303
Directing the Abstract Base Factory	304
Lab 5.3 – Factory Method Pattern	305
Other Techniques	306
Directory of Named Objects	307
JNDI – Example	308
Dependency Injection Frameworks	309

Session Review	310
Session 6: Using Composition and Inheritance Effectively	311
Session Objectives	312
Inheritance and Composition Pros and Cons	313
Composition	314
Delegation	315
Benefits of Composition	316
Issues with Composition	317
HAS-A versus USES	318
Composition vs. Inheritance – Which Is Better?	319
The First Principles of OO	320
Strategy Pattern	321
Encapsulating Variation in an Algorithm	322
Strategy Pattern in UML	323
Alternatives to Strategy	324
A Different Kind of Variation	325
Implementing Strategy – Making the Decision	326
Implementing Strategy – Uniform API	327
Implementing Strategy – Getting the Data	328
Example – Comparator	329
Lab 6.1 – Strategy Pattern	330
Lab 6.2 – More Complex Strategy [Optional Homework Lab]	331
Decorator Pattern	332
Changing Responsibilities at Runtime	333
Decorator Pattern – Runtime View	334
Decorator Pattern in UML	335
Decorator Example – Class Diagram	336
Decorator Example – Base Classes	337
Decorator Example – Concrete Component	338
Decorator Implementations	339
Decorator Implementations	340
Assembling the Pieces	341
Lab 6.3 – Decorators	342
Façade and Other Patterns	343
- Façade Pattern -	344
- Proxy Pattern -	345
- Template Method Pattern -	346
Template Method – Example	347
Session Review	348
Session 7: Inner Classes	349
Session Objectives	350
Overview and Motivation	351
Quick Note on Terms	352
Motivation for Inner Classes – Encapsulation	353
Motivation for Inner Classes – Convenience	354
Life Before Inner Classes – Mini-Lab	355
Inner Classes in Action – Mini-Lab	356
Why Use Inner Classes?	357
Why Use Inner Classes?	358
Inner Classes – Rules	359
Think in Objects	360

Defining and Using Inner Classes	361
- Member-Level Classes -	362
Visibility of Member-Level Classes	363
- Method-Local Classes -	364
- Anonymous Classes -	365
Anonymous Member-Level Class	366
Anonymous Class as an Argument	367
Static Nested Classes	368
Nested Class ≠ Inner Class	369
Using Nested Classes – Example	370
Nested Interfaces, Nested Enums	371
Lab 7.1 – Inner and Nested Classes	372
Session Review	373
Session 8: Annotations	374
Session Objectives	375
Overview	376
What Are Annotations?	377
Why Are They Used?	378
Annotated Class Definition – Example	379
But How Does It Work?	380
Using Annotations	381
How to Use Annotations	382
Target and Retention	383
Annotation Parameters	384
Parameter Shortcuts – <code>value</code>	385
Parameter Shortcuts – Array-Valued Types	386
Lab 8.1 – Using Annotations	387
Writing Custom Annotations	388
Define Your Annotation's Intent and "API"	389
Custom Annotation Syntax	390
Using the Meta-Annotations	391
Using the Custom Annotation – Example	392
Lab 8.2 – Writing Custom Annotations	393
Session Review	394
Session 9: Reflection	395
Session Objectives	396
Overview and API	397
What Is Reflection?	398
The Class Called <code>Class</code>	399
Reflection API – Overview	400
Obtaining the <code>Class</code> Object	401
Everything Has a Class – Even Primitives	402
Inspecting a Class via the <code>Class</code> Object	403
Lab 9.1 – Inspecting with Reflection	404
Working with Objects Reflectively	405
Working with Objects Reflectively	406
Creating Instances	407
Invoking Methods	408
Setting Field Values	409
Private Reflection in Java 11	410

Lab 9.2 – Working Reflectively [Optional Demo Lab]	411
Session Review	412
Session 10: Lambda Expressions	413
Session Objectives	414
Overview	415
Motivation: Common Actions Are Verbose	416
Too Much Window Dressing	417
Introducing Lambda Expressions	418
Functional Interfaces and Lambdas	419
Lambdas Occur in a Target Context	420
Relationship to Functional Interfaces	421
Using Lambda Expressions	422
Lambda Expression Syntax – Overview	423
Lambda Compatibility	424
What about the Parameter Types?	425
Lambda Expression Syntax – Details	426
Lambda Expression Syntax – Details	427
Type Inference – Using <code>var</code> with Lambdas	428
What's the Big Deal? Where Do I Use Them?	429
Lab 10.1 – Getting Started with Lambdas	430
Method References	431
Lambdas Can Leverage Existing Code	432
Lambdas and Local Variables	433
Variable Capture in Lambdas – Example	434
Method References	435
Types of Method References	436
Recipes to Use Method References	437
Refactoring Lambdas into Method References	438
Lab 10.2 – Method References	439
The Other Side of the Method Call	440
Using a Functional Interface Type – Example	441
Session Review	442
Session 11: Streams	443
Session Objectives	444
Overview	445
Collections Are Great...for What They Do	446
Collections Have Shortcomings	447
Analogy: Data vs. DBMS	448
Analogy: Collections = Data, Streams = DBMS	449
Streams – Before and After	450
What Are Streams?	451
Anatomy of a Stream – Details	452
Anatomy of a Stream – Illustrated	453
Streams vs. Collections	454
Stream Example – a First Look	455
Understanding the Stream API	456
Stream API – Overview	457
Using the Stream API – Illustrated	458
Chained Method Calls – Illustrated	459
Keys to Understanding the API	460
Keys to Understanding the API	461

Java API Functional Interfaces	462
Functional Interface: <code>Predicate<T></code>	463
Functional Interface: <code>Comparator<T></code>	464
Functional Interface: <code>Function<T, R></code>	465
Stream Processing	466
<code>filter(Predicate)</code>	467
<code>sorted()</code> <code>sorted(Comparator)</code>	468
<code>Comparator.comparing()</code> – Details	469
Comparator Methods Added in Java 8	470
Comparator Chaining – Example	471
Lab 11.1 – Filtering and Sorting	472
<code>map(Function)</code>	473
<code>map(Function)</code> – Example	474
Functional Interface: <code>Consumer<T></code>	475
<code>peek(Consumer)</code>	476
"Trimming" Operations of Stream	477
Terminal Operations of Stream	478
Terminal <code>void</code> Operations – Example	479
Terminal Reduction Operations	480
Terminal Reduction Operations – Example	481
Existence Operations of Stream	482
Finder Operations of Stream	483
Statistics Operations	484
Lab 11.2 – Advanced Stream Processing	485
Collectors	486
Role and Capabilities of Collectors	487
Collectors – Our Approach	488
<code>Collectors.toList()</code> and <code>Collectors.toSet()</code>	489
Functional Interface: <code>Supplier<T></code>	490
<code>Collectors.toCollection()</code>	491
<code>Stream.collect(Collector)</code> Method – Details	492
Collector Interface – Details	493
Determining a Collector's Product	494
Determining a Collector's Product	495
Lab 11.3 – Getting Started with Collectors	496
Partitioning and Grouping – Overview	497
Partitioning Collectors	498
Grouping Collectors	499
Lab 11.4 – Partitioning and Grouping	500
Session Review	501
Session 12: Introduction to Modules	502
Session Objectives	503
When to Move to Java Modules?	504
Motivation and Overview	505
Issues before Java Modules	506
A Natural Progression of Organization	507
Package-Level Encapsulation	508
Modular Encapsulation – Not in Java 8	509
Internal Use Only – Case Study	510
Conceptual Model vs. Runtime Model	511
Conceptual Model vs. Runtime Model	512
Classpath Hell	513
Adopters of the Module Paradigm	514

Module Graphs	515
Introducing Java Modules	516
Modular Encapsulation – Realized in Java 9	517
Module Name vs. Package Name	518
Modules and Directory Structure	519
Modular Projects and the Module Path	520
Modules Can Be Controversial	521
Lab 12.1 – First Module Project	522
Types of Modules	523
Types of Modules – Defined	524
Application Modules	525
Platform Modules	526
Automatic Modules	527
Automatic Modules – Why?	528
Unnamed Module	529
Unnamed Module – Why?	530
Modular JDK	531
Modular JDK – Different, Yet Compatible	532
Java SE vs. JDK – Not the Same!	533
Foundation Modules – <code>java.base</code> and <code>java.se</code>	534
Benefits of the Modular JDK	535
Our Approach	536
Initial Work	537
Ongoing Study	538
Lab 12.2 – Simple Migration	539
Session 13: Working with Modules	540
Session Objectives	541
Defining and Using Modules	542
Modular Applications	543
Lab 13.1 – Modular Application	544
Module Graphs	545
Root Module and Module Graph	546
Module Graph in a Real library – Derby JDBC	547
Module Graph in the Java Platform – <code>java.se</code>	548
Transitive Dependencies	549
Aggregator Modules	550
Qualified Exports	551
Opening up Deep Reflection	552
Modular Encapsulation – Summary	553
Services	554
Services Overview	555
Services – UML Diagram	556
ServiceLoader before Modules	557
ServiceLoader with Modules	558
Domain Class – Example	559
Service Interface – Example	560
Service Provider – Example	561
Client – Example	562
Lab 13.2 – Services [Demo Lab]	563
Compatibility and Migration	564
Optional / Gradual Adoption	565

Issues with Private (Deep) Reflection	566
Migration Issues	567
Automatic and Unnamed Modules – Revisited	568
Recommended Migration Path	569
Recommended Migration Path – Illustrated	570
Split Packages	571
Shouldn't I Just Ditch the Classpath?	572
Command Line Directives	573
Lab 13.3 – Migration [Optional Homework Lab]	574
Conclusion	575
Conclusion – It's Not Conclusive!	576
IDE Support	577
Build Tools and Dependencies	578
What about Module Versions?	579
Library Support	580
Modules and Testing	581
Modular Java EE	582
To Module or Not to Module?	583
To Module or Not to Module?	584
Adoption is slow – Mid 2019 Stats	585
Module Resources	586
Lab 13.4 – Multi-Module Maven Project [Optional Demo Lab]	587
Recap	588
Recap of What We've Done	589
Recap of What We've Done	590
Resources	591



Intermediate/Advanced Java (11+)

Version 20200320

Copyright © LearningPatterns Inc. All rights reserved.

1

Notes:

- ◆ Version 20200320

Course Overview

- ◆ 5-day course to deepen your understanding of the Java language, its core APIs, and OO design / development
 - **Goal:** help you become a better Java and OO developer
- ◆ **Assumption:** you already know Java (at least the basics)
- ◆ It covers the following high-level areas:
 - **UML** and **design patterns**
 - Using **composition and inheritance** effectively
 - Advanced language features: **inner classes, writing generic classes, custom annotations, reflection**
 - Modern language features: **lambda expressions and functional programming, Stream API, Java modules, type inference**
 - **Unit Testing:** JUnit used extensively throughout the course

Introduction

2

Notes:

Course Overview

- ◆ This is a **very full** intermediate-advanced level course
 - Builds on basic Java knowledge to use it productively
 - Goes well beyond the basics
- ◆ Be prepared to work hard and learn a great deal!
- ◆ The course contains numerous hands-on labs
 - They exercise all the important concepts discussed
 - Lab solutions for the course are provided
- ◆ The course is suitable for **Java 11** and later

Introduction

3

Notes:

- ◆ You may find the course quite challenging, and that's okay.
 - The primary goal of the course is to help you become a solid Java / OO developer. Java is so big now, a 5-day Intro course + a few months experience just isn't enough anymore.
 - The course pushes you to work hard, and we don't go into every little detail like in an Intro course – we're beyond that at this point.
- ◆ There will be a lot of hands-on work in the lab exercises, and you may see some things in the starter code that are new to you.
 - These are things that you will see every day as a Java developer, and you need to start getting comfortable with them.
 - By the end of the course, you will be a much more confident Java developer.

Course Agenda

- ◆ Preface: **Java State of the Union**

- ◆ Session 1: **Review – Basics**

See **Review Sessions – Guidelines** immediately following this agenda

- ◆ Session 2: **Review – Inheritance and Interfaces**

- Includes a primer on **UML** (used throughout the course)

- ◆ Session 3: **JUnit**

- Starting here, most of the labs will be run as JUnit tests

- ◆ Session 4: **Collections and Generics**

- ◆ Session 5: **Techniques of Object Creation**

- Includes an overview of **design patterns**

- Several patterns are introduced in the course

Introduction

4

Notes:

Course Agenda

- ◆ Session 6: **Using Composition and Inheritance Effectively**
- ◆ Session 7: **Inner Classes**
- ◆ Session 8: **Annotations**
- ◆ Session 9: **Reflection**
- ◆ Session 10: **Lambda Expressions**
- ◆ Session 11: **Streams**
- ◆ Session 12: **Introduction to Modules**
- ◆ Session 13: **Working with Modules**

By the end of these two sections, you **will** understand Functional Programming in Java

Introduction

5

Notes:

Review Sessions – Guidelines

- ◆ Review sessions 1 and 2 are included for a few reasons:
 - Reinforce and deepen understanding of Java core concepts
 - Provide a level set, to get ready for the remaining sessions
 - Introduce features you may not have had exposure to
 - Even if you've been doing Java for a while
- ◆ Content of the review sessions is **two-fold**:
 - Accelerated, yet thorough hands-on review of Java fundamentals
 - Attention given to OO design and implementation principles
 - Coverage of **newer language features** (Java 8 and later)
- ◆ Spend class time on the most relevant areas in these reviews
 - Perhaps the entire review sessions are covered in full
 - Or you focus more on the new features material

see notes

Introduction

6

Notes:

- ◆ This is a **very full** 5-day course, which was designed to be used in a variety of contexts:
 - Developers currently working in Java and want to advance their knowledge.
 - Seasoned OO developers from other languages that want to learn Java at a more advanced level than an intro course would provide.
- ◆ This will affect how the course is best delivered – specifically, the content, pace, and labs.
 - Based on student backgrounds and experience level, and interests and goals, the course can be conducted so that more class time is spent in the areas of most value.
 - Spend your time wisely!

Typographic Conventions

- ◆ Code in the text uses a fixed-width font
`Catalog catalog = new CatalogImpl()`
- ◆ Code fragments are the same, e.g., `catalog.findById(4)`
- ◆ We **bold/color** text for emphasis
- ◆ New terms are often introduced in ***bold italics***
 - Filenames and paths are also in italics, e.g., *Catalog.java*
- ◆ Notes are indicated with a superscript number ⁽¹⁾ or a **star ***
- ◆ Longer code examples appear in a separate code box (below)

```
package com.javatunes.catalog;

public class CatalogImpl implements Catalog {
    public MusicItem findById(long id) {
        ...
    }
}
```

Introduction

7

Notes:

⁽¹⁾ If we had additional information about a particular item in the slide, it would appear here in the notes, with the appropriate superscript.

- ◆ We might also include related information that pertains more generally to the slide.

Labs

**Lab**

- ◆ The course has numerous hands-on lab exercises
 - Many use types from a fictional case study called **JavaTunes**
 - An online music store
 - The lab instructions are separate from the main manual
- ◆ Setup files are provided with skeleton code for the labs
 - Students add code focused on the topic at hand
 - There is a solution zip with completed lab code
- ◆ Lab slides have an icon in the upper right corner of the slide
 - The end of a lab is marked with a stop sign



Introduction

8

Notes:



Preface: Java State of the Union

Brief overview of broader changes and issues in the **post-Java 8 world**

It is a non-goal of this section to list every detail, nor to identify which release each item applies to

Notes:

New Java Release Cycle

- ◆ New "major" version every 6 months – **March** and **September**
 - Started with Java 9 in Sep 2017
 - Two kinds of releases, detailed below
- ◆ **Feature Release**
 - Implementation enhancements, possible new language features
 - Not intended for long-term use
 - **Superseded** after 6 months, with release of next version
- ◆ **Long-Term-Support (LTS) Release**
 - This is what you're accustomed to working with – up to Java 8
 - Plan is to mark a release as LTS every **3 years**
- ◆ Details in Oracle Java SE Support Roadmap [see notes]

Preface: Java State of the Union

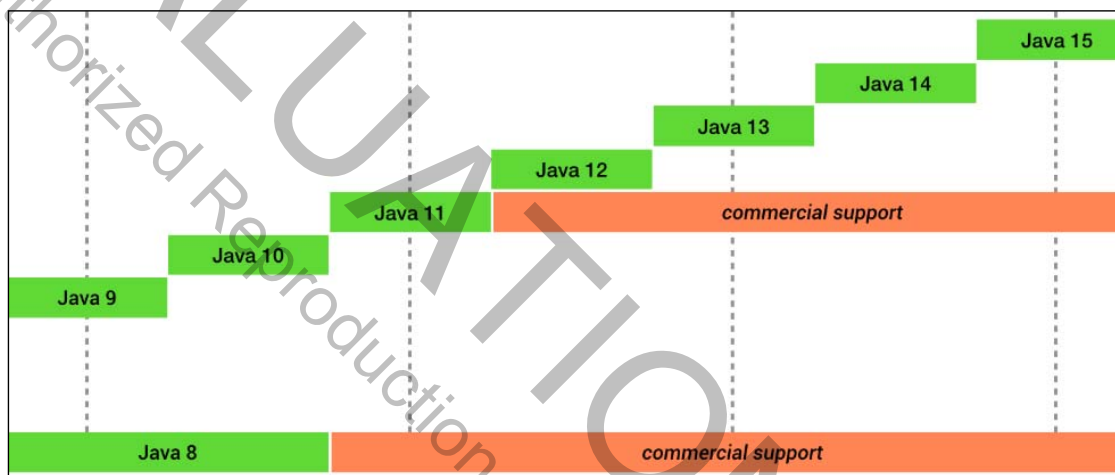
10

Notes:

- ◆ From the **Oracle Java SE Support Roadmap** at <http://oracle.com/technetwork/java/java-se-support-roadmap.html>.
 - For product releases after Java SE 8, Oracle will designate a release, every three years, as a Long-Term-Support (LTS) release. Java SE 11 is an LTS release. For the purposes of Oracle Premier Support, non-LTS releases are considered a cumulative set of implementation enhancements of the most recent LTS release. Once a new feature release is made available, any previous non-LTS release will be considered superseded. For example, Java SE 9 was a non-LTS release and immediately superseded by Java SE 10 (also non-LTS), Java SE 10 in turn is immediately superseded by Java SE 11.
- ◆ This is a **very** important document, packed with detail (and myriad footnotes). If you are responsible for maintenance and updates at your organization, you really need to read it in its entirety.

Versions and Support – Illustrated

- ◆ Except for LTS releases, these are really just 6-month "technology previews"
 - Official term is "feature release"
- ◆ LTS releases span several feature releases
 - **Java 17** is the next planned LTS, in September 2021



Preface: Java State of the Union

Notes:

- ◆ This release schedule offers some nice benefits, in that updates are more frequent, and predictable. (Though the versioning and non-LTS vs. LTS releases will take some getting used to.)
 - As a consequence, with more frequent releases, it can be more difficult to quickly find the applicable documentation for a specific release. Release Notes and other docs for all releases can be found by simply starting at your JDK installation, and opening `<jdk-install>/README.html`, then click and browse from there.

- ◆ Here is a detailed history of releases and updates since Java 9. Look it over closely – see how it works?
- ◆ Given the new release schedule, the next LTS release after 11 will be 17 LTS, slated for September 2021.

Non-LTS and LTS releases are more frequent now			
– 9.0	Sep 2017	March and September each year	one month later, coincides with below Jan-Apr-Jul-Oct update schedule ←
9.0.1	Oct 2017		
9.0.2	Jan 2018		
– 10.0	Mar 2018	JDK 9 is dead	
10.0.1	Apr 2018		
10.0.2	Jul 2018		
– 11.0 LTS	Sep 2018	next LTS in 3 years , JDK 10 is dead	
11.0.1	Oct 2018		
11.0.2	Jan 2019	– 12.0	Mar 2019
11.0.3	Apr 2019	12.0.1	Apr 2019
11.0.4	Jul 2019	12.0.2	Jul 2019
11.0.5	Oct 2019	– 13.0	Sep 2019
etc., until next LTS		13.0.1	Oct 2019

What's In

◆ Java Modules *

JPMS = Java Platform Module System

- All code now resides in a module (of some type)
 - Including all your Java 8 code written years ago (more on this later)
- We now have *modulepath* in addition to the classpath

◆ JShell – REPL shell for quick-coding and running scripts

◆ HTTP Client – replaces outdated HttpURLConnection

◆ Reactive Streams – non-blocking pub-sub stream processing

◆ Other enhancements – see notes

- Collection factory methods *
- Multi-release JARs (MR-JARs)
- Type inference – Java has a **var** now *
- Enhanced switch construct *

* covered in the course

Preface: Java State of the Union

12

Notes:

- ◆ The Java Platform Module System (JPMS), initially called Project Jigsaw, appeared in Java 9.
 - It is the most significant new feature, and has the potential to affect nearly every aspect of Java development and runtime execution – or not... As we'll see later, even if you "opt out" of creating your own modules, you still need to know how it all works.
- ◆ Additional miscellaneous capabilities include **Process API** updates, and a new **Stack-Walking API** for working with exception traces.
- ◆ There have also been platform runtime and monitoring changes.
 - G1 is the default collector + new Epsilon GC (no-op), and ZGC.
 - The **Epsilon** garbage collector is a **no-op**, particularly relevant for short-lived programs that don't benefit from the cost of GC. <http://openjdk.java.net/jeps/318>.
 - **ZGC** is a low-latency concurrent collector targeted at very large heap applications. <http://openjdk.java.net/jeps/333>.
 - **Java Flight Recorder** has been donated to open source, no longer requiring a commercial license. <http://openjdk.java.net/jeps/328>.

What's Out

- ◆ Java 9 introduced new "attributes" of deprecation
 - **since** and **forRemoval** can now be included in `@Deprecated`
 - And note that items ARE now being marked for actual removal
 - Time to finally take this seriously and change your old code...

```
@Deprecated(since="1.2", forRemoval=true)
```

- ◆ Java 11 took this one step further, **physically removing them**
 - Overlapping APIs from Java EE (now Jakarta EE)
 - Client-side UI items
 - **Applets** deprecated in Java 9 and removed in 11
 - **Java Web Start** removed in Java 11, with no replacement
 - **JavaFX** extracted out to OpenJFX – <http://openjfx.io>
- ◆ No more JRE
 - Instead, build a (smaller!) custom runtime image with **jlink**

Preface: Java State of the Union

13

Notes:

- ◆ The new JDK tool **jdeprscan** can be run on your code, and it will tell you all the places you're using deprecated items. (So now you really have no excuse...)
- ◆ Several overlapping APIs from Java EE have been physically removed in 11. The most likely to affect you is the removal of **JAXB** and **JAX-WS** – the `javax.xml.bind` packages and `javax.xml.ws` packages. <http://openjdk.java.net/jeps/320>.
 - If you need them, just provide the libraries (JARs) as for any other external dependency.
- ◆ Note: with all the client-side UI pieces being removed or extracted out, Swing and AWT are **still in**.
 - Don't need 'em in your application? See "No more JRE."
- ◆ There have also been changes to the Java runtime and monitoring tools included with the JDK.
 - **No more JRE** – instead, build a (smaller!) custom runtime image with **jlink**.
 - **Java VisualVM** (`jvisualvm`) and **Java Mission Control** (`jmc`) have been extracted out as separate downloads now, and **Java Flight Recorder** has been donated to open source.
 - This was mostly done to align Oracle JDK with OpenJDK. As of Java 11, these two are now "functionally equivalent, with cosmetic differences."

What We'll Cover

- ◆ We can't do everything in 5 days (!)
 - We've chosen the features that are most relevant for this course
- ◆ To dive into all the major new features, there is a 2-day **Java Modules and New Features** course that includes:
 - Advanced topics in Java modules
 - Multi-release JARs (MR-JARs)
 - HTTP Client
 - JShell
 - Reactive programming in Java
 - Creating custom runtime images with `jdklink`

Notes:

Thinking in Java 9+

- ◆ JDK command-line tools have evolved over time
 - More options / flags than ever before
 - Many of these can now be expressed in several forms:

```
$ java -cp
$ java -classpath
$ java --class-path // this is the new form
$ java -p
$ java --module-path // this is the new form
```

- **NOTE:** since some of the new options offer only the new form, we will be favoring this one (in the slide examples and labs)
- ◆ New compiler option **--release**
 - Replaces legacy `-source` and `-target` flags

Notes:

- ◆ The new `--release` option would be used as: `javac --release <version> ...`
 - Source code is checked for language features not available in `<version>`.
 - Also checks that you're not using newer APIs not available in `<version>`.
 - Generates correct bytecode for the target release.
- ◆ Previously, most people would just use an older JDK for compilation and testing, and not worry about `-source` and `-target`.
 - The new `--release` option provides comprehensive source code verification and bytecode generation to support older Java runtimes.
 - This means you no longer need to have a previous JDK installed to support earlier releases.



Session 1: Review Basics

- Java Environment
- Classes and Objects
- Packages, Enums, Arrays
- Exceptions
- Date and Time API
- New Language Features

Session 1: Review – Basics

16

Notes:

Session Objectives

- ◆ Review Java landscape, development / runtime environment
- ◆ Review basic constructs: classes & objects, packages, enums
 - Build on this: equals() and hashCode(), Comparable and Comparator (and their contracts), static fields and methods
 - Reaffirm principles: data encapsulation, cohesion
- ◆ Review basic constructs: arrays, flow of control, exceptions
- ◆ Discuss important new features in Java 8 and beyond
 - Date/Time API, Java modules, type inference, switch expressions
- ◆ **Goal:** warm up with some basic Java and OO review
 - **And start to build on that**
 - This "review" may yield lots of new understanding! (a good thing)

Session 1: Review – Basics

17

Notes:



Java Environment

Java Environment

- Classes and Objects
- Packages, Enums, Arrays
- Exceptions
- Date and Time API
- New Language Features

Notes:

Java – a Language and a Platform

- ◆ Source code (.java) is compiled into bytecode (.class)
 - Generally in an IDE such as Eclipse, IntelliJ, etc.
- ◆ Bytecode executes on the Java Virtual Machine (JVM)
 - Which in turn, executes on the target operating system
- ◆ Three main platforms:
 - **Java SE** – compiler, runtime (JVM), and standard API libraries
 - String, System, JButton, Collection – thousands of classes
 - Plus other utilities – jar, javap, etc., found in **JAVA_HOME/bin**
 - **Java EE** – specs and APIs for back end enterprise systems
 - Implementations provided by app servers – JBoss, WebLogic, etc.
 - Transferred to Eclipse Foundation, and renamed **Jakarta EE**
 - We'll still use the term Java EE in the course
 - **Java ME** – for smaller devices
 - Subsumed by Android for (non-Apple) smartphones

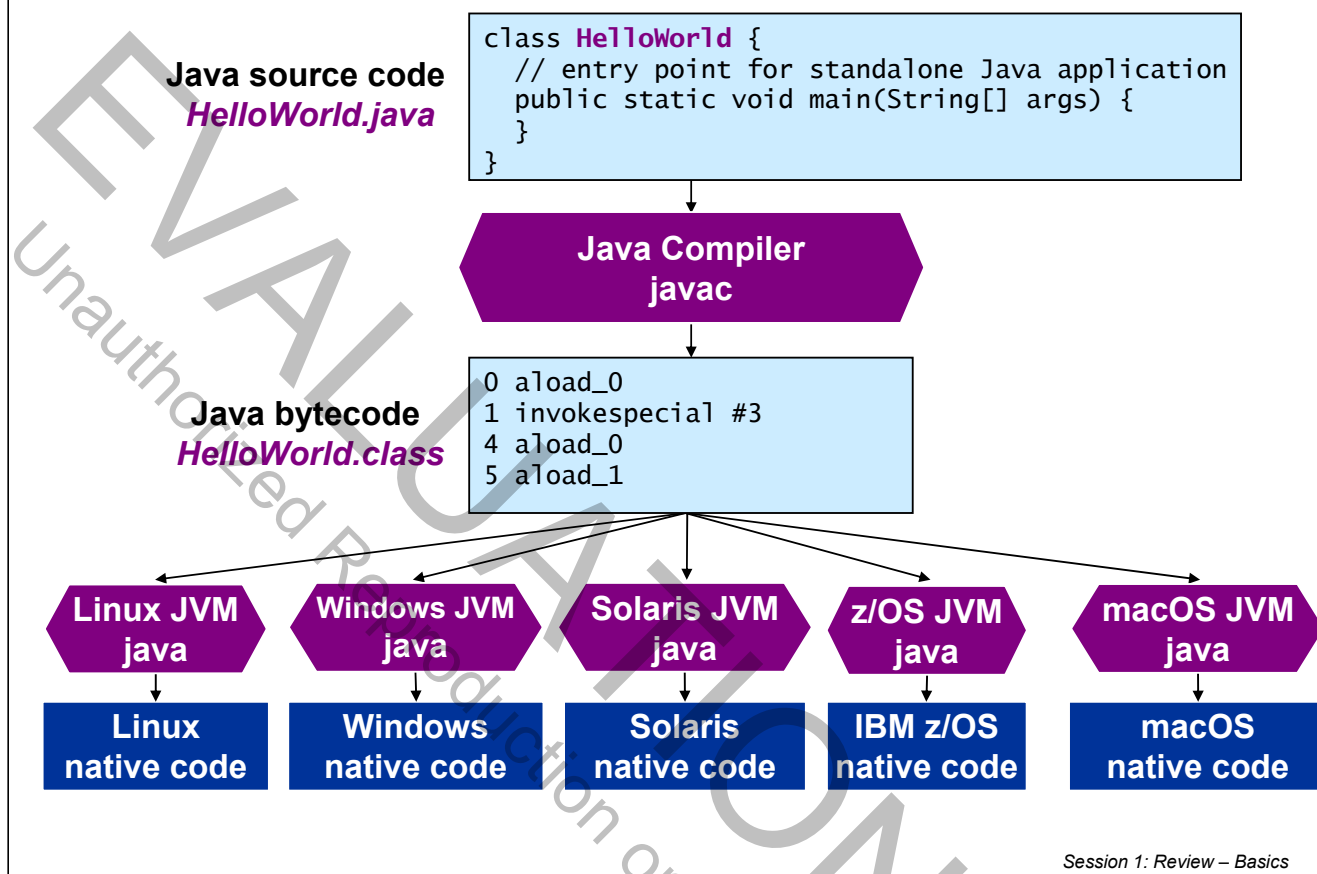
Session 1: Review – Basics

19

Notes:

- ◆ The standard Java Core API library is part of every Java SE installation. This includes thousands of classes from the java and javax packages – e.g., java.lang, java.util, javax.naming, javax.xml.stream, etc.
 - Over time, many "extension" packages, named "javax.something," were moved into the Core API library, but retained their original "javax" package name for compatibility.
 - As one example, the Swing UI library, originally java.swing, changed names to javax.swing and was pulled out as a "standard extension." Later, it was brought back into Core, but retained its "javax" package name.
 - Don't get hung up on this – any package that's included in the "Java Platform, Standard Edition API Specification" (a.k.a. "Java Core API") is part of Core, regardless of its package name.
- ◆ Java ME is targeted for "smaller devices," before the huge popularity of smartphones.
 - Google's Android system is based on Java ME, but needed more, so it incorporated elements of Java SE, as well as its own vendor-provided libraries.
 - Java ME is still provided and maintained by Oracle.
- ◆ As of Java EE 8, this platform was transferred to the Eclipse foundation, and renamed Jakarta EE
 - See: https://www.eclipse.org/community/eclipse_newsletter/2019/august/jakartaee8.php

Lifecycle of a Java Program – Illustrated

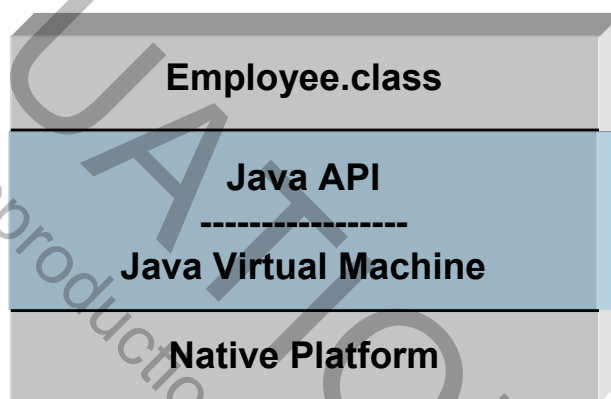


Notes:

- ◆ JDKs for Windows, Solaris, Linux, and macOS are available from Oracle. For an IBM z/OS mainframe or AIX system, it's from IBM, HP-UX from Hewlett Packard, etc.
 - Historically, the JVM for any given platform has generally been provided by that respective vendor. Note, however, that there are exceptions to this:
 - IBM provides a JDK for Windows (through Java 8). Its Rational Application Developer IDE bundles the IBM JDK for Windows, which is required to run RAD. IBM also bundles its JDK with the WebSphere Application Server on Windows, and requires it to run WebSphere.
 - Nowadays, you have a choice between the **Oracle JDK** and **OpenJDK**, on several platforms.
 - The open-source OpenJDK is an increasingly popular choice for today's main platforms: Windows, Solaris, Linux, macOS, and AIX. <http://adoptopenjdk.net>.
- ◆ Remember, the JDK and JVM are platform-specific, but the compiled bytecode (.class files) is portable across all JVMs.
 - For example, you can compile on a Windows JDK and run on a Linux JVM.

Java Language vs. Java Runtime

- ◆ Java platform executes bytecode, regardless of its source
 - Software-only platform comprised of the Java Core API and JVM
 - Source code can be written in other languages, too!
 - Groovy, Kotlin, Clojure, Scala, Jython, JRuby, others
 - These compilers create `.class` files – just like **javac**
 - These languages have full access to the Java API



Session 1: Review – Basics

21

Notes:

- ◆ The Java platform has become just as important as the Java language itself.
 - There is a huge win in this – you can write code in JVM languages like Groovy and Kotlin, taking advantage of each one's features, yet still retain access to the full Java API and the portability of the Java runtime environment.
 - From the JVM's perspective, a `.class` file is a `.class` file – bytecode is bytecode.

Applications, Deployments, and Libraries

- ◆ Application types:
 - **Standalone**: uses own JVM, bootstrapped via `main()` method
 - **Server-side**: runs in Java EE app server, uses app server JVM
 - Deployed into server as web app (WAR) or enterprise app (EAR)
- ◆ Packaging / deployment types:
 - **JAR**: standalone application or Java library
 - Contains `.class` files and resource files ⁽¹⁾
 - **WAR**: web application
 - Like a JAR, but has additional structure and Java EE XML files
 - **EAR**: generalized server-side application package
 - EAR = WAR(s) + JAR(s) together in one file
- ◆ Myriad 3rd party libraries also available – generally as JARs

Session 1: Review – Basics

22

Notes:

⁽¹⁾ Resource files are simply all files that are not `.class` files. They include `.properties` files, images, XML files, etc. – any files that might be needed at runtime.

- ◆ JAR / WAR / EAR files are just zip files with `.jar`, `.war`, and `.ear` extensions, respectively.
 - They are all created via the JDK `jar` utility or your IDE or build tool.
 - We go into the details of WARs and EARs in server-side development courses.
- ◆ There are countless 3rd party libraries available in Java, many of them open-source and free to use and distribute with your applications.
 - They run the gamut, from sophisticated enterprise libraries like Spring and Hibernate, to simpler utility libraries that allow you do such things as read an Excel spreadsheet from Java.
 - They are almost always packaged as JARs.



Lab 1.1 – Getting Started

Notes:



Intermediate/Advanced Java (11+) Lab Manual – Eclipse

Version 20200320

Copyright © LearningPatterns Inc. All rights reserved.

1

Notes:

- ◆ Version 20200320

Release Level



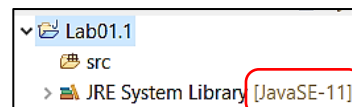
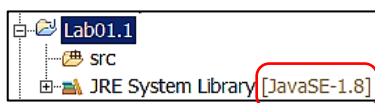
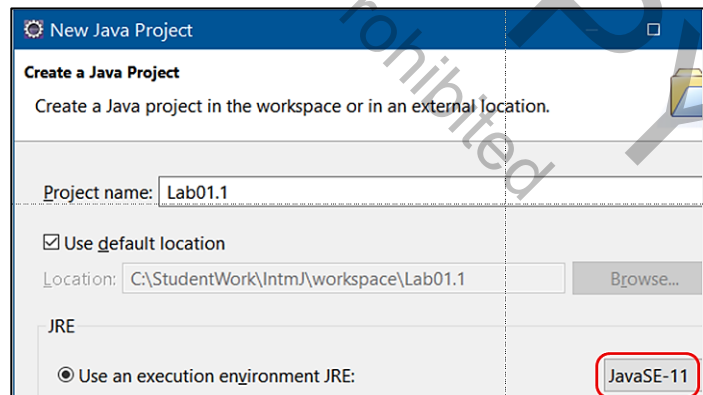
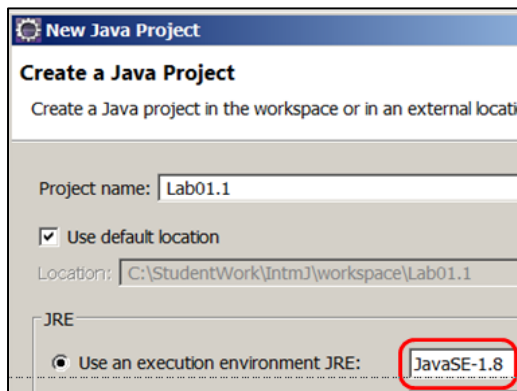
- ◆ This manual contains instructions for creating and running the labs using the following software packages:
 - **Java 11**
 - **Eclipse IDE for Enterprise Java Developers – 2019-12 or later**
 - Labs were developed and tested on **2019-12**
Using this version or later is recommended
 - Version 2018-12 was the first release to fully support Java 11
- ◆ Instructions are geared towards Windows-based operating systems
 - You should easily be able to adapt to other environments, e.g., macOS
- ◆ **See notes** regarding screenshots

Notes:

- ◆ The course was originally written for Java 8 and the labs developed with Eclipse Oxygen/4.7.
 - We have not updated every screenshot, because in many cases the differences are minor and not important to completing the lab.

Eclipse 4.7 Classic Theme with Java 8
Windows 7 Classic Theme

Eclipse 2019-12 Classic Theme with Java 11
Windows 10 Aero Theme





Lab 1.1 – Getting Started

Lab 1.1 – Getting Started

3

Notes:

Lab Synopsis

A blue starburst shape containing the word "Lab" in white, italicized font.

◆ Overview:

- Set up our lab environment and the Eclipse IDE
- Write the canonical HelloWorld application in the IDE, and run it there
- Run the application from the command line
- Create a JAR file for our application and run it using the JAR
- Extract and bookmark the Java Core API Javadoc

◆ Builds on previous labs: none

◆ Approximate time: 15-20 minutes

Lab 1.1 – Getting Started

4

Notes:

Information Content and Task Content

Lab

- ◆ **Informational content** is presented in the normal way – the same as in the student manual
 - Like these bullets at the top of the page
- ◆ **Tasks** you need to do are in a "Tasks to Perform" box

Tasks to Perform

- ◆ Look at these instructions, and notice the different look of the box
 - Make a note of how it looks, as future labs will use this format
- ◆ Make sure that you have **Java 11 installed**
 - Likely in a directory such as **C:\Program Files\Java\jdk-11.0.6** ⁽¹⁾
 - If not, you'll need to install it – it can be downloaded from:
<http://oracle.com/technetwork/java/javase/downloads>
- ◆ OK – now **get out your setup files**, we're ready to start working

Lab 1.1 – Getting Started

5

Notes:

⁽¹⁾ On macOS, the JDK is typically installed under `/Library/Java/JavaVirtualMachines/jdk-11.0.6`.

- ◆ The setup files may be on your workstation, or they may be provided by your instructor.

Extract the Lab Setup Zip File

Lab

- ◆ To set up the labs, you'll need the course setup zip file
 - It has a name like: **LabSetup_IntmJ_20200330.zip**
- ◆ Our base working directory for this course will be **C:\StudentWork\IntmJ**
 - Gets created when we extract the lab setup zip
 - Includes a directory structure and files needed in the labs
 - All instructions assume that this zip file is extracted to **C:**
If you choose a different directory, please adjust accordingly

Tasks to Perform

- ◆ Unzip the lab setup file to **C:**
 - Creates the **StudentWork\IntmJ** directory structure, containing files that you will need for doing the labs

Lab 1.1 – Getting Started

6

Notes:

- ◆ The setup files may be on your workstation, or they may be provided by your instructor.

Java Core API

Lab

- ◆ There are **thousands** of classes in the Java Core API
 - It's **critical** that you know how to read their APIs and use them
- ◆ Available online, but nice to have it "installed" as part of your JDK
 - Online: <http://docs.oracle.com/en/java/javase/11/docs/api>
 - Installed: download docs zip and extract into JDK installation directory
 - **Note**: separate download from the JDK installer executable

Tasks to Perform

- ◆ On the filesystem, find `C:\StudentWork\Intm\Resources`
 - Note the file `jdk-11-api-docs.zip` (exact filename may be different)
 - Extract this zip file to your Java installation directory – likely in a directory such as `C:\Program Files\Java\jdk-11.0.6` ⁽¹⁾
- ◆ Open `<jdk>/docs/api/index.html` in a browser – **and bookmark it**
 - See notes regarding differences from previous versions of Javadoc ⁽²⁾

Lab 1.1 – Getting Started

7

Notes:

- ◆ It's nice to have the documentation installed locally, so you're not dependent on an Internet connection. If you're using a later update than 11.0.6, these docs still apply.
- ◆ We'll sometimes use `<jdk>` as an abbreviation for the JDK installation directory.

⁽¹⁾ On a Windows system, you may not be able to extract the zip directly into the Java installation directory, because it's under the special "Program Files" directory.

- As a workaround, you can extract to a temp directory, or do an "extract to here," then **move** (don't copy) the resulting **docs** directory to your JDK installation directory.

⁽²⁾ Looks familiar, but the front page is a list of **modules** now, not packages.

- To get to the packages, click on module `java.base` and then it's business as usual.
- Note the **Search** field upper right!



The Eclipse Platform

 Lab

- ◆ **Eclipse** (www.eclipse.org) is an open source platform for building integrated development environments (IDEs)
 - Used mainly for Java development – can be extended via plugins and used in other areas (e.g., C# programming)
- ◆ Originally developed by IBM
 - Released into open source
 - IBM's RAD product line is built on top of Eclipse
- ◆ Eclipse products have two fundamental layers:
 - **Workspace**: files, packages, projects, resource connections, configuration properties – all stored on the **filesystem**
 - **Workbench**: editors, views, and perspectives – the **UI** you work with
- ◆ We will set up the workspace and workbench, then do our lab

Lab 1.1 – Getting Started

8

Notes:

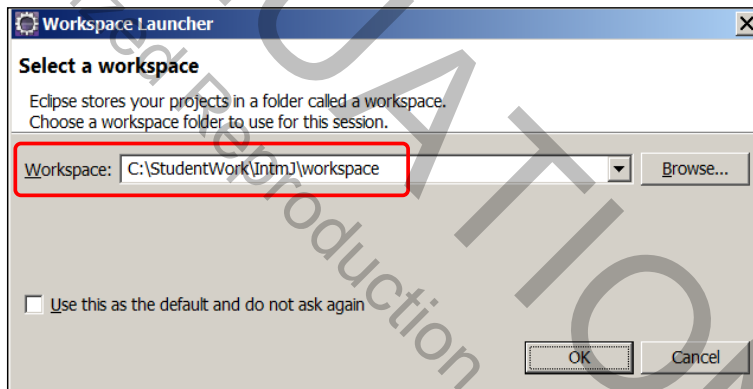
- ◆ The workbench sits on top of the workspace and provides visual artifacts (the UI) that allow you to access and manipulate the underlying workspace resources.

Getting Started with Eclipse

Lab

Tasks to Perform

- ◆ Make sure you have **Eclipse installed** – likely in **C:\eclipse**
 - If not, you'll need to install it – see instructions in notes
- ◆ **Launch Eclipse:** go to **C:\eclipse** and run **eclipse.exe**
 - A dialog box should appear prompting for a workspace location ⁽¹⁾
 - Set the workspace location to **C:\StudentWork\IntmJ\workspace** ⁽²⁾
 - If a different default workspace location is shown, change it
 - **Don't import any projects** – just open the workspace ⁽³⁾



Lab 1.1 – Getting Started

9

Notes:

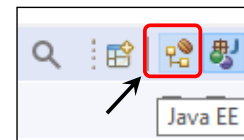
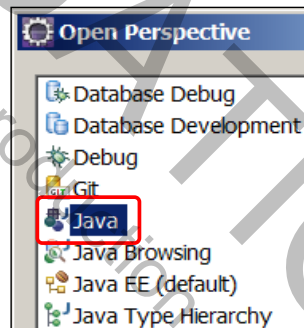
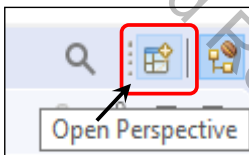
- (1) If the workspace dialog box does **not** appear, that means that Eclipse was previously launched on your system, a workspace directory was chosen, and the "Use this as the default..." box was checked.
 - In this case, you should change workspace to that shown above via:
File → Switch Workspace → Other... This will open the dialog box.
 - (2) **NOTE: you must use our provided workspace directory**, as we've supplied starter code and other files for some of the labs here.
 - (3) You should not do anything but open Eclipse using the correct workspace here.
 - In particular, you should not import any project folders you see. We'll explain how to use these all correctly later, and it's important to wait until then.
- ◆ If Eclipse was installed elsewhere, adjust the path to the Eclipse executable accordingly.
 - ◆ You can also put a shortcut on your desktop to start Eclipse.
 - ◆ If you need to download Eclipse, go to <http://www.eclipse.org/downloads/packages>.
 - **Ignore** the large banner "Try the Eclipse Installer" and instead use the links under the heading "Eclipse IDE for Enterprise Java Developers." These links are for the zip file installations.
 - Save the zip file to your computer, and unzip it. The easiest location to unzip it to is C:\, but another location is fine as long as you can get to it to run the **eclipse.exe** executable.

Workbench and Java Perspective



Tasks to Perform

- ◆ Close the Welcome screen (click the X on its tab)
- ◆ **Open a Java Perspective**
 - Click the Perspective icon at the top right of the Workbench and select **Java** (see below left and below center)
 - See notes about perspective icons
- ◆ **Close the Java EE perspective** by right-clicking its icon at the top right of the Workbench and selecting Close (see below right)
 - The Java EE perspective is the default for the Eclipse Java EE version



Lab 1.1 – Getting Started

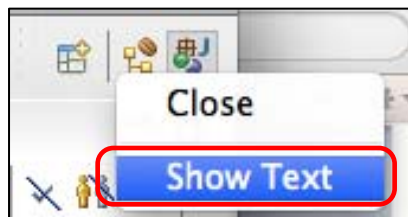
10

Notes:

- ◆ **NOTE:** the perspective switcher does not show the perspective name by default. To change this, right-click on any perspective icon and **Show Text** – see examples below.



without labels



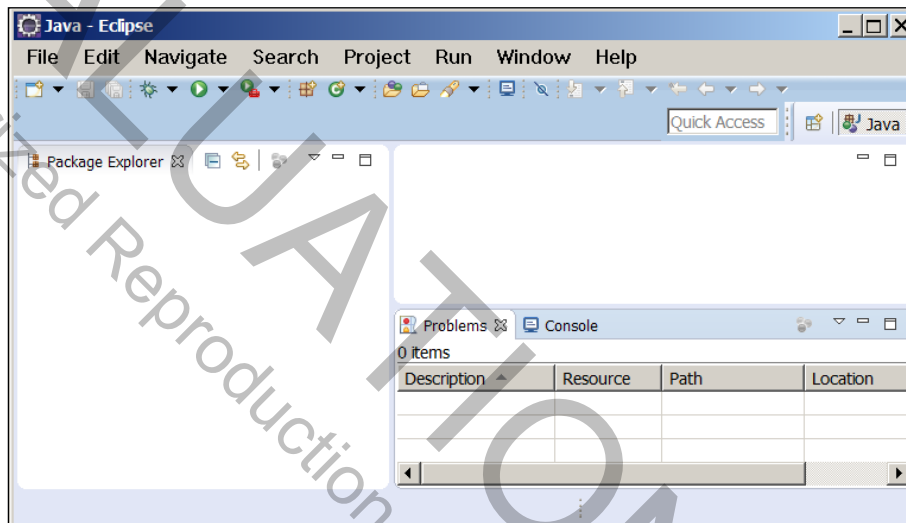
with labels

Customize the Perspective

Lab

Tasks to Perform

- ◆ Unclutter the perspective by closing some views
 - Close the Task List and Outline views (click on the X)
- ◆ Open the Console view: **Window → Show View → Console**
- ◆ You can save these changes to the perspective (see notes)

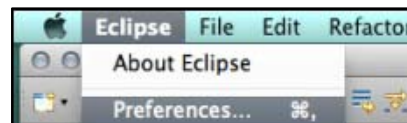


Lab 1.1 – Getting Started

11

Notes:

- ◆ An Eclipse **perspective** is simply a collection of **views**, shown as tabbed panes in the Workbench.
 - A perspective defines which views are included, where they are positioned, and what their relative sizes are in the Workbench.
- ◆ Eclipse has several predefined perspectives, which can be used as-is, or customized to suit your needs / preferences. There's a good chance you'll want to tweak the ones you're using.
 - To customize a perspective, open a predefined one, e.g., the Java perspective, close the views you don't want, open any views you do want, reposition and/or resize them, and then save the changes via **Window → Perspective → Save Perspective As...**
 - TIP: to preserve the factory-shipped perspective and create a brand new perspective based on it, use a different name, e.g., "Java [YI]", where "[YI]" are Your Initials in square brackets.
 - Window → Perspective → Save Perspective As... → Java [YI].
- ◆ Your instructor may also recommend additional customizations, which s/he will go over with you.
 - There are myriad settings, all available via **Window → Preferences**.
 - On macOS, the Preferences panel is reached via **Eclipse → Preferences**.

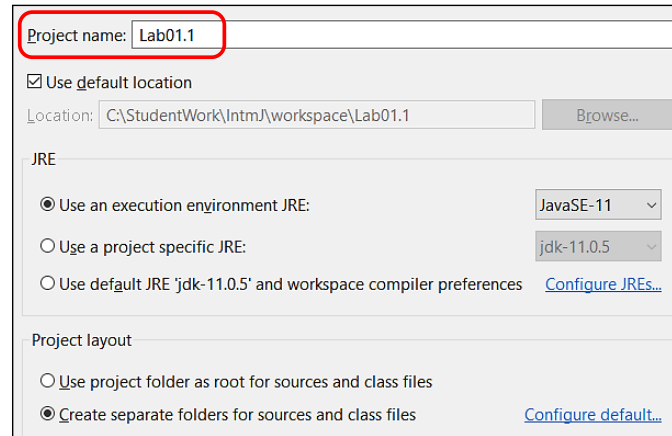


Create Project

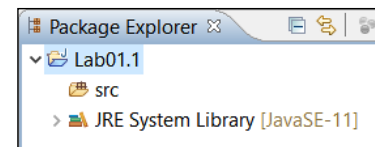
Lab

Tasks to Perform

- ◆ Create a new **Java** project
 - **File** → **New** → **Java Project**
 - Name it **Lab01.1**
- ◆ Note the remaining defaults
 - Location: **workspace/Lab01.1**
 - Separate folders for source and class files (best practice)



- ◆ **Finish**
 - When prompted for module name, click **Don't Create**
 - **NOTE: always** choose Don't Create until told otherwise – more on this later *
- ◆ Your new project in the Package Explorer view:

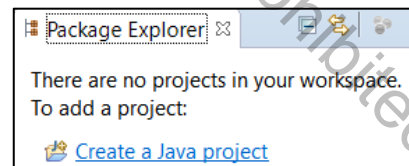


Lab 1.1 – Getting Started

12

Notes:

- ◆ There are multiple ways to create a new project.
 - File → New → ...
 - Ctrl+N.
 - Click on the "New" wizard icon on left side of the toolbar.
 - Right click anywhere in the Package Explorer view, select New ...
 - Use the link "Create a Java project" in the Package Explorer view.
 - Note that this link is only present when you have no projects yet.



- ◆ It's a best practice to create separate folders for source and class files.
 - Eclipse puts the *.java* files in the lab's *src* directory, and the *.class* files in the lab's *bin* directory.
 - This is the default structure for Java projects in Eclipse.

- * We'll work with the new Java module system later.
 - For now, just **never** create a module for your projects.

Write and Run HelloWorld

Lab

Tasks to Perform

- ◆ Create package named **com.hello**
 - Right-click on **src** and choose **New → Package**
 - All classes should be in a package, even this simple one
- ◆ Create class **HelloWorld** in the **com.hello** package
 - Right-click on package name and choose **New → Class**
 - Complete the class as shown

```
package com.hello;

public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

- ◆ Run it: right-click on **HelloWorld.java** and choose **Run As → Java Application**

Lab 1.1 – Getting Started

13

Notes:

Set JAVA_HOME

Lab

- ◆ To use the JDK tools on the operating system command line, you should set the **JAVA_HOME** environment variable
 - Then add **JAVA_HOME/bin** to your **PATH**
 - **May already be done** – check for it first

Tasks to Perform

- ◆ Find where your JDK is installed
 - Likely in a directory such as **C:\Program Files\Java\jdk-11.0.6**
- ◆ Open the Windows **System** Control Panel
 - Choose "Advanced system settings"
 - In the dialog, select "Advanced" tab and click "Environment Variables" button
 - Create or modify the **JAVA_HOME** variable to point to your JDK installation
 - Add **%JAVA_HOME%\bin** to the **front** of the **PATH** variable
 - See your instructor if you need help



Lab 1.1 – Getting Started

14

Notes:

- ◆ Setting JAVA_HOME may have been specified in your Classroom Setup instructions. It's always a good idea to have it, even if you do "everything" from the IDE, because at some point you'll need to run something or use one of the JDK tools on the command line.

Run the Application Externally


 Lab

- ◆ You'll use the **java** command line tool to run the app
 - Will need to specify the **classpath** – where java looks for classes
 - Best practice: use the **-classpath** flag (see notes for details)

```
$ java -classpath <path-to-classes;path-to-jars> YourClassName
```

Tasks to Perform

- ◆ Open a command prompt to
C:\StudentWork\IntmJ\workspace\Lab01.1
 - The *HelloWorld.class* file is in a package directory under *Lab01.1/bin*
- ◆ Run `HelloWorld` from the prompt – use **fully qualified classname**

```
C:\StudentWork\...\Lab01.1>java -classpath bin com.hello>HelloWorld
Hello world!
```

- If you get a "command not found" error, there's a problem with your `JAVA_HOME` and/or `PATH`

Lab 1.1 – Getting Started

15

Notes:

- ◆ The java command line tool is the JVM launcher.
 - It requires a main-class to run (a class with a `main()` method).


```
$ java YourClassWithMain
```
 - For a usage banner, just type `java` at the command prompt and press [Enter].
- ◆ We're using a relative path for the classpath (relative to *Lab01.1*) – the **bin** directory.
 - That directory contains our "loose" *.class* file – *HelloWorld.class*, physically stored in the appropriate package subdirectory *com/hello*.
 - The runtime classpath can be set two ways:
 - `CLASSPATH` environment variable (global to the operating system).
 - The `-classpath` flag specified at runtime (which overrides any `CLASSPATH` setting).
 - If the `CLASSPATH` environment variable is not set, and the `-classpath` flag is not provided, the classpath defaults to the current directory (`.`).

Create Application JAR File

Lab

- ◆ Java software is generally shipped in JAR files
 - We'll create one here, and use it to run our application

Tasks to Perform

- ◆ In Eclipse, right-click on *src* and choose **Export...**
 - Expand the "Java" category and choose "JAR file"
 - Next
- ◆ In the next dialog, specify destination
 - Easiest to just type it in: *Lab01.1\hello.jar*
 - File is created relative to *workspace* directory
 - Press Next **twice**, to the dialog named "JAR Manifest Specification" (see notes)
- ◆ Specify that the JAR has a main-class
 - **Finish**

Select the class of the application entry point:

Main class:

Select an export destination:

type filter text

- General
- EJB
- Install
- Java
 - JAR file
 - Javadoc
 - Runnable JAR file

JAR File Specification

The export destination will be relative to your workspace.

Select the resources to export:

- Lab01.1

.classpa
 .project

Export generated class files and resources
 Export all output folders for checked projects
 Export Java source files and resources
 Export refactorings for checked projects. [Select refacto...](#)

Select the export destination:

JAR file:

Lab 1.1 – Getting Started

16

Notes:

- ◆ Full dialog screenshot at right.
- ◆ Details on the main-class specification:
 - Every JAR file has a **MANIFEST.MF** file, stored in the **META-INF** directory of the JAR.
 - It's a simple text file with name-value pairs.
 - Various things can be specified, using standardized manifest file attributes.
 - We're specifying that the JAR has a main-class, and we're providing its name.
 - The resulting manifest file will look like this:
Manifest-Version: 1.0
Main-Class: com.hello.HelloWorld
 - This allows us to "run the JAR" as an application. We do this on the next page.

JAR Export

JAR Manifest Specification

Customize the manifest file for the JAR file.

Specify the manifest:

Generate the manifest file

Save the manifest in the workspace
 Use the saved manifest in the generated JAR description file

Manifest file:

Use existing manifest from workspace

Manifest file:

Seal contents:

Seal the JAR

Seal some packages Nothing sealed

Select the class of the application entry point:

Main class:

Run the Application Using the JAR

Lab

Tasks to Perform

- ◆ From your command prompt in *workspace/Lab01.1*:

```
$ java -classpath hello.jar com.hello.HelloWorld  
Hello world!
```

- Here, `-classpath` specifies the location of the JAR file
- As before, we're using a relative path (relative to *Lab01.1*), and we need to specify the fully qualified name of the main-class

- ◆ Now run it as an "executable JAR" via the `-jar` flag

```
$ java -jar hello.jar  
Hello world!
```

- Relies on the JAR having a **Main-Class** attribute in its manifest file
 - We specified this during JAR file creation
 - That's why you don't have to specify the classname of the main-class when you run it

Lab 1.1 – Getting Started

17

Notes:

Create a Launch Script [Optional]

Lab

- ◆ Many applications ship with a launch script
 - End user simply executes the script, which sets up the classpath, and may additionally configure other application-specific properties

Tasks to Perform

- ◆ Using a text editor, create a *run-hello.cmd* file in the *Lab01.1* directory
 - Here's a very simple one:

```
@echo off
rem     Executes the HelloWorld application
java -jar hello.jar
```

- ◆ Try it out:

```
$ run-hello
Hello world!
```

Lab 1.1 – Getting Started

18

Notes:

Lab Summary

 Lab

- ◆ This "simplest of all labs" still contained quite a bit of work(!)
- ◆ In this course, we are exposing you to some of the things you **need to know** as a more experienced Java developer – in this case:
 - JAVA_HOME
 - Classpath
 - Creating and using JAR files
 - Application launch scripts
- ◆ These are things you will repeatedly encounter



Lab 1.1 – Getting Started

19

Notes:

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited



7400 E. Orchard Road, Suite 1450 N
Greenwood Village, Colorado 80111
Ph: 303-302-5280
www.ITCourseware.com